Building secure embedded kernels with the Think architecture.

Christophe Rippert, Jean-Bernard Stefani

Building secure embedded kernels with the THINK architecture

Christophe Rippert, Jean-Bernard Stefani
SARDES project, LSR-IMAG laboratory, CNRS-INPG-INRIA-UJF
INRIA Rhône-Alpes, 655 av. de l'Europe, Montbonnot 38334 St Ismier Cedex, France
{Christophe.Rippert, Jean-Bernard.Stefani}@inria.fr

Abstract

We present in this paper the security features of THINK, an object-oriented architecture dedicated to build customized operating system kernels. The THINK architecture is composed of an object-oriented software framework including a trader, and a library of system abstractions programmed as components. We show how to use this architecture to build secure and efficient kernels for embedded systems. Policy-neutral security is achieved by providing elementary tools that can be used by the system programmer to build a system resistant to denial of service attacks and incorporating data access control. An example of such a secure system is given by detailing how to ensure component isolation with a elementary software-based memory isolation tool.

1 Introduction

Ubiquitous computing is based on highly-heterogeneous platforms whose only common point is often that they have very restricted capabilities compared to standard workstations. Thus, these embedded systems require even more than other platforms a customized operating system with drivers optimized for the underlying hardware and no unnecessary system abstraction loaded in the operating system kernel. On the other hand, writing a system from scratch is a tedious and difficult task that increases production costs and requires expert programmers. The THINK architecture is a solution to this problem as it provides an object-oriented framework and a complete library of operating system abstractions for the programmer to use when he builds his system. One of the main advantages of the THINK architecture is the flexibility it provides to the system programmer, since all components are optional and can be loaded and unloaded dynamically. However, flexibility must not be ensured at the expense of security and quality of service, especially in embedded systems where resources are scare.

We present in this paper the work we have conducted in the THINK architecture to provide a secure framework. We first present the THINK architecture and its framework. Then we describe the software-based memory isolation mechanism we have implemented, and present how it can be used to provide component isolation. Finally we list some related works before concluding.

2 The Think architecture

2.1 The software framework

The distributed systems architecture THINK is a platform for the development of distributed operating systems kernels. The goal of the THINK architecture is to ease the development of efficient, flexible, and secure operating systems. THINK provides the system programmer with interfaces that reify the underlying hardware, and optional system abstractions proposed as libraries. The development of a kernel with THINK is made easier by its object-oriented framework, since in THINK, all resources (both hardware and software) are considered as objects. These ob-

jects export interfaces which define their behaviour and make them accessible to other objects. Each interface has a name in a given naming context, and is linked to other interfaces by bindings. A binding is essentially a communication channel between two objects. These bindings can take many forms, as simple as the association between a variable name and its value, or more complex like a binding over a network between two objects on different machines. Bindings are created by dedicated objects called binding factories, whose main function (i.e. creating bindings) can be freely extended to enforce a chosen behaviour. Finally, objects can be grouped in domains according to a common property (e.g. security domains, fault-domains, etc).

All those concepts are presented in the minimal software framework detailed in Figure 1. Java is used in THINK as the interface description language. The `Top` interface is the greatest element in the THINK type lattice, the common type from which all interfaces derive. The `Name` interface is the common type for all names in THINK. The method `getDefaultNC` returns the current naming context and the method `toString` provides a serialized form of the name. The `NamingContext` interface is the common type for all naming contexts in THINK. Its method `toName` deserializes a name known as a String. The method `export` provides a name for a given interface. As a side effect, it also creates a binding between the returned name and the given interface. The `BindingFactory` interface is the common type for all binding factories in THINK. The method `bind` creates a binding between the calling object and the object which name is given to the method.

The THINK framework also includes a simple trader component. This trader exports two methods, `register` and `lookup`, whose prototypes are given in Figure 2. The `register` method permits to publish the reference (i.e. an instance of `Name` in Think) to a given service under a symbolic name which eases its localisation and abstracts its implementation. For example, a memory manager can be registered under the symbolic name "`mem_manager`" which other components will use to locate it. Moreover, the THINK architecture supports the registration of multiple services under the same symbolic name. For example, the symbolic name "`mem_manager`" can represent two different memory manager, one for flat memory and one for paged memory. The `lookup` method is used to find the reference to a service known by its symbolic name. The second parameter, `hints`, is used to specify which service is wanted in case of multiple registration under the same symbolic name. For example, in the previous example of two memory managers, the programmer could call `lookup(``mem_manager'', ``flat'')` to obtain the `Name` of the flat memory manager.

```
interface Top {
}

interface Name {
  NamingContext getDefaultNC();
  String toString();
}

interface NamingContext {
  Name toName(String name);
  Name export(Top itf);
}

interface BindingFactory {
  Top bind(Name name);
}
```

Figure 1: The core software framework in THINK.

```
interface Trader {
  void register(Name name,
                String symbName);
  Name lookup(String symbName,
              String hints);
}
```

Figure 2: The Trader component interface.

A more detailed presentation of THINK framework can be found in [1].

2.2 Think for embedded systems

The THINK architecture is especially interesting for embedded systems. These systems are usually composed of very specific hardware with limited performances compared to a standard workstation, therefore requiring the operating system to make optimal use of the underlying hardware. Unnecessary abstractions should not even be present in the kernel so as not to waste any memory, and drivers should be optimized for the specific hardware. This is exactly the philosophy of the THINK architecture which provides a library of optional abstractions to be used by the system programmer as suits him best. Since each entity is a component, and no component is mandatory, the programmer can compose his system as needed, and replace any provided component by his own, thus building a customized system optimized for the target platform. Moreover, the intrinsic dynamicity of the THINK architecture makes it ideal for building context-aware systems. Since components can be loaded and unloaded dynamically, the system can evolve according to its environment, and the trader component permits to hide the actual implementation of a service, thus making the evolution of the system due to variations of its environment transparent to the client. All these properties make the THINK architecture a valuable tool to build context-aware operating systems for ubiquitous computing.

3 Protection in the Think architecture

The THINK architecture permits to build flexible and adaptable systems. However, security is a critical issue in a modern operating system and should not be compromised by flexibility. We are working to provide in the THINK architecture the tools necessary for the system programmer to build a secure system which offers protections of data and resistance to denial of service (DoS) attacks. These tools must be policy-neutral so as not to compromise the flexibility of the system. We present here one of these elementary tools, a software-based memory isolation mechanism, and how it can be used to provide component isolation in the THINK framework.

3.1 A software-based memory isolation mechanism

The software-based memory isolation mechanism implemented in the THINK architecture uses a modified version of the software-based fault isolation algorithm presented in [2]. The principle of the algorithm is to parse the code of a process at creation time and to replace each memory access (i.e. load, store and branch instructions) by a branch instruction which points to generated code. This generated code simply checks that the destination address of the memory access is in a allowed zone before executing it. In case of an unauthorized access, the code throws an exception. We detail this algorithm with a simple example in Figure 3. The original code of this example consists of a simple memory access, loading into register `r1` an integer located at the address computed by adding 8 to the value of register `r2`. When the process is created, this load instruction is replaced by a branch to the generated code. To simplify the example, we consider that there is only one memory area in which the process is allowed to read, and that this area is defined by its lower and upper addresses stored in registers `r15` and `r16`. So the generated code simply check that the destination address (computed in register `r14`) is between the bounds in `r15` and `r16`.

```
Original_code:
  lwz 1,8(2) // r1 := [r2 + 8]

Modified_code:
  ba Generated_Code // goto Generated_code

Generated_code:
  add 14,2,8  // r14 := r2 + 8
  tw 8,14,15  // if r14 < r15 then trap
  tw 16,14,16 // if r14 > r16 then trap
  lwz 1,8(2)  // r1 := [r2 + 8]
  ba Original_code + 4 // return
```

Figure 3: Code modification and generation by the software-based isolation algorithm.

3.2 Component isolation using software-based memory isolation

The memory isolation algorithm presented above can easily be used to provide efficient and flexible component isolation in the THINK architecture. Component isolation is necessary to prevent the following misbehaviors:

- a component reading or modifying another component's private data;
- a component calling a method it is not allowed to call (i.e. an internal method which the server component has not exported in its interface);
- a component calling a method without using the calling procedure imposed by the framework. This could be the behaviour of a process trying to short-circuit an accounting mechanism implemented in a method to launch a denial-of-service attack for example.

Software-based isolation as we have implemented it in the THINK architecture allows the definition of memory areas with different permissions. For example, an area into which a component can branch but not read or write can be seen as an execution-only area, just as some hardware isolation mechanisms permit execute-only segments to be defined for application code. However, in the case of hardware isolation, permissions are usually globally fixed (i.e. if a segment is tagged as read-only, then no process in the system can write to that segment, be it a system or an application process[1]). Our software-based isolation mechanism, on the other hand, permits permissions to be defined on a per-process basis: since permission checking is done by inserting code in the process itself, the same memory area can be tagged as read-only for one process and execution-only for another one for example, thus achieving a complete flexibility in the isolation of components.

[1]Some architectures (e.g. the Intel ia32 architecture) permit privilege levels to be defined for processes of different classes (e.g. kernel, system services, applications, ...), but this usually remains very restricted (the Intel ia32 architecture defines only 4 different privilege levels for example)

We present how to use the THINK framework to isolate components by detailing a simple example. Considering two components, a client component and a server component, we want to ensure that the client component which wants to call the `alloc` method exported by the server component does it using the advocated way through the framework and not directly by forging a pointer to the method entry point for example. Moreover, if this `alloc` method is a front-end to the real allocation method (that could be the case for example if we want to keep track of the amount of resource allocated to each process), we want to prevent the client component from short-circuiting this front-end by calling directly the underlying procedure.

Using the software-based isolation mechanism, we define 3 memory areas with different permissions, from the client component point of view:

- Zone 1 is the memory area containing the client component itself, and is therefore accessible to it in Read, Write, and eXecute modes.
- Zone 2 is the memory area containing the framework well-known entities, such as the trader and the local binding factory, and can be accessed by the client component only in eXecute mode (i.e. the component can call methods from these entities but cannot either read or modify data or code in this area).
- Zone 3 is the rest of the system memory, which is not accessible at all from the client component.

In this isolation scheme, the client component cannot call the `alloc` method on the server component since the server is in a zone that the client cannot access at all. Since we want to allow this call, we extend the role of the local binding factory: when the client requests the creation of a binding, the binding factory generates a stub, which is responsible for forwarding the call issued by the client and the corresponding return from the server method. The address of the stub is returned to the client which will use it instead of the real `alloc` method address. Since the stub is in an execution-only zone of memory from the client point of view, the client component can call it but not

modify it, therefore forcing it to (indirectly) call the `alloc` in the advocated way. The generated stub is simply a method, the `stubMethod`, whose only effect is to call the `alloc` method of the server component.

The figure 4 illustrates this example. The sequence of actions necessary to call the `alloc` method is detailled below:

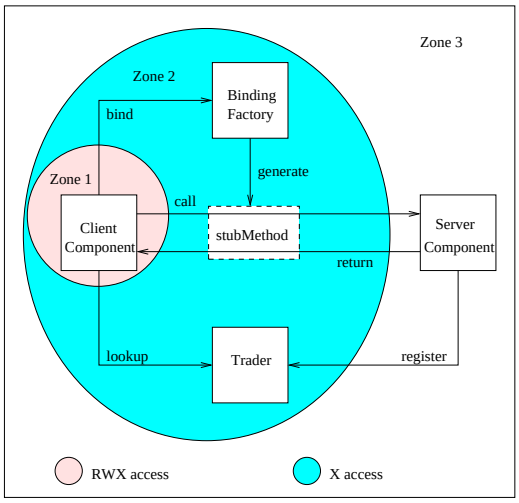

Figure 4: Software framework for component isolation.

1. The server component registers itself in the trader, thus exporting the `alloc` method through its interface.
2. The client component uses the `lookup` method of the trader to find the service provided by the server component.
3. The client component requests the creation of a binding between itself and the server component to the local binding factory.
4. The binding factory, after checking that the client component has the right to call the `alloc` method, generates the stub which will forward the call to the server component and the return from it. The address of this stub is returned to the client component as a result of the bind method.
5. The client component calls the `stubMethod` which then calls the `alloc` method of the server component.
6. Once the `alloc` method is over, the execution goes back to the `stubMethod` which ends and returns to the client component.

Compared to inter-process communication over hardware isolation, software-based memory isolation has proven its efficiency (see [2] section 5). In our architecture, we monitored the cost of an absolute branch instruction with software-based memory isolation and found that the increase of the execution time was only of +16.67%. We compared this inter-process call with an optimised LRPC [3] implemented over hardware isolation in THINK and found that the LRPC is more than 25 times slower than our mechanism.

4 Related work

Compared to the OSKit [4] framework and set of libraries, THINK provides a better flexibility since all components are independent. The KaffeOS [5] project proposes some techniques to preserve quality of service in a Java environment, using the type safety properties of the language. Similarly, the SPIN [6] extensible operating system uses the properties of the Modula-3 language to permit the safe binding of modules in the operating system. In THINK, we aim to remain independent of the component development language. In the DTOS [7] project, policy-neutral security is enforced by way of security servers, which check that inter-component calls are allowed. This requires a modification of the component source code, whereas in THINK, binding factories can make security checks, and the binary code of the component is modified by the software-based memory isolation mechanism without needing any modification of the component source code. The Scout/Escort [8] project focuses on protection against denial of service attacks by defining the I/O path abstraction. However, this does not take into account the resources allocated in the operating system kernel, whereas in THINK, we aim toward a global view of the resources

which enable the system to associate each allocated resource with the benefiting user. This point of view is close to the resource containers abstraction [9], although this work has been conducted in monolithic kernels whereas in THINK we advocate a more modular architecture. The exo-kernel [10] advocates the same philosophy of a minimalist kernel, although it does not propose an object framework for the components as in THINK.

5 Conclusion and future work

As we have seen in this paper, the THINK architecture can be used to build flexible and secure operating systems for specific architectures like embedded systems. The software-based memory isolation mechanism presented here is one of the policy-neutral elementary tools provided in the THINK architecture. Coupled with THINK component-based framework and high-level abstractions like a policy-aware security manager, these tools make the THINK architecture a valuable base for the system programmer to build a secure and DoS attack resistant system. Some work remains to be done in the specification and implementation of the security manager, the component dedicated to manage the security policy defined by the system programmer. This component must be able to use the policy-neutral elementary tools to enforce the chosen policy. An important part of this work is the definition of the language used to define security policy and we believe that Domain Specific Languages [11] might prove to be the appropriate tools for that purpose. With that secure and flexible framework, the THINK architecture shall prove to be a fitting tool for the building of customised kernels, thus bringing a high level of security and flexibility in the ubiquitous world.

References

[1] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall and Gilles Muller. THINK: A Software Framework for Component-based Operating System Kernels. In Proceedings of the USENIX Annual Technical Conference, 2002.

[2] Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham. Efficient Software-Based Fault Isolation. In Proceedings of the ACM SIGOPS'1993.

[3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy. Lightweight Remote Procedure Call. In ACM Transactions on Computer Systems, Vol. 8, No. 1, February 1990, pages 37-55.

[4] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, 1997.

[5] Godmar Back, Wilson C. Hsieh, Jay Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation, 2000.

[6] Przemyslaw Pardyak, Brian N. Bershad. Dynamic Bindings for an Extensible System. In Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, 1996.

[7] Duane Olawsky, Todd Fine, Edward Schneider, Ray Spencer. Developing and Using a "Policy Neutral" Access Control Policy. In Proceedings of the New Security Paradigms Workshop, 1996.

[8] Olivier Spatscheck, Larry L. Peterson. Defending Against Denial of Service Attacks in Scout. In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, 1999.

[9] Gaurav Banga, Peter Druschel, Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, 1999.

[10] Dawson R. Engler, M. Frans Kasshoek, James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995.

[11] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages – An Annotated Bibliography. ACM SIGPLAN Notices, 2000.